1 Overview

Your objective in this homework is to make code self-contained and repeatable using methods. You will do this in the context of solving several disconnected problems that have no overarching story. Sorry.

Breaking code up into methods has several benefits: it enables large problems to be decomposed into smaller, mind-sized bytes; methods with a distinct purpose are easier to test than gangly spans of code; and because of the scope boundaries of methods, data becomes more insulated and less likely to get accidentally overwritten or inappropriately referenced. Additionally, methods jive with a basic human desire to make things that have lasting value. Like a family recipe, methods stand the test of time and get passed around.

2 Warning

When one is first learning about methods, there’s a great temptation to acquire the parameters from the user and print out the method’s return value inside the method:

```java
public void triple(int n) {
    Scanner in = new Scanner(System.in);
    System.out.println("What is n? ");
    n = in.nextInt();
    System.out.println(n + n + n);
}
```

Don’t do this. Methods generally should be independent of input from and output to the user. Where the input comes from and where the output goes to should be decided by the caller of the method—not the method itself. Our method `triple` is most reusable and testable when we write it like this:

```java
public int triple(int n) {
    return n + n + n;
}
```

Written properly, we can feed it data from many different sources:

```java
triple(in.nextInt());
triple(generator.nextInt());
triple(17);
```

Additionally, we can embed the return value in many different contexts:

```java
int thrice = triple(17);
System.out.println(triple(2));
int b = a + triple(5);
```
3 Requirements

Complete the four classes described below. Place all classes in package hw2. Make all methods static.

3.1 Main

Write a class Main with a main method, which you are encouraged to use to test your code. Nothing in particular is required of it, but it must exist.

3.2 NumberUtilities

Write class NumberUtilities with the following:

1. Method round10, which accepts one parameter: a number to round, of type double. It returns the number rounded to the nearest multiple of 10 and converted to an int.
   - NumberUtilities.round10(17.6) \(\rightarrow\) 20
   - NumberUtilities.round10(-103.3) \(\rightarrow\) -100

2. Method getGameCount, which accepts one parameter: a number of rounds in a tournament, of type int. It returns as an int the number of games that must be played to determine the tournament’s winner. For example, a tournament with only 1 round of play is decided with just 1 game between teams A and B. A tournament with 2 rounds is decided with 3 games: A vs. B, C vs. D, and the championship game. For example:
   - NumberUtilities.getGameCount(1) \(\rightarrow\) 1
   - NumberUtilities.getGameCount(2) \(\rightarrow\) 3
   - NumberUtilities.getGameCount(3) \(\rightarrow\) 7

3. Method getFraction, which accepts one parameter: a number of type double. It returns as a double the fractional portion of the number, the value of the digits after the decimal point. For example:
   - NumberUtilities.getFraction(3.14159) \(\rightarrow\) 0.14159
   - NumberUtilities.getFraction(-100.01) \(\rightarrow\) 0.01

3.3 WebUtilities

Write class WebUtilities with the following:

1. Method getImageLink, which accepts three parameters in this order:
   a) a path to an image file, of type String
   b) a width of type int
   c) a height of type int
It returns as a String the HTML that presents the image at the given width and height and also as a clickable link. For example, `getImageLink("alpaca.png", 1024, 768) → <a href="alpaca.png"><img src="alpaca.png" width="1024" height="768"></a>.

2. Method `getHost`, which accepts one parameter: a URL of type String. For this problem, we will define the parts of a URL in the following way: `protocol://host/path`. This method returns the host portion of the URL, which lies between the 2 and 3 forward slashes. Assume at least three slashes will always be present. Do not assume that the host always begins at the same index. The protocols of different URLs may vary in length.

3. Method `getTitle`, which accepts one parameter: some HTML source of type String. It returns as a String the text between the opening and closing title tags. For example, `getTitle("<html><head><title>Trees I Have Hugged</title></head><body>...</body></html>") → Trees I Have Hugged. Assume that only one title tag appears in the HTML and it is formatted as in the example.

4. Method `invertHexColor`, which accepts one parameter: an HTML color of type String. Assume the color is a hexadecimal triplet of the form #rrggbb, where rr, gg, and bb are respectively the red, green, and blue intensities of a color. Each is represented as a two-digit hexadecimal number in [0, 255]. This method returns as a String the inverse (or complement) of that color, also as a hexadecimal HTML color. If the red intensity is 50, its inverse is 255 − 50 = 205. For example:
   - `WebUtilities.invertHexColor("#000000") → #ffffff`
   - `WebUtilities.invertHexColor("#ff99cc") → #006633`
   - `WebUtilities.invertHexColor("#147acf") → #eb8530`

You won’t be able to perform arithmetic on the color as long as it’s locked up in a String. Use `Integer.parseInt` with a base of 16 to convert hexadecimal characters to a decimal int. To convert an int intensity back to hexadecimal, use `String.format`—which behaves like `printf` but returns rather than outputs—with format specifier `%02x`. This will convert the corresponding int parameter to its lowercase hexadecimal equivalent, with leading zeroes as necessary to pad it out to two digits. For example: `String.format("%02x", 15) → 0f`.

3.4 `BrailleUtilities`

Write class `BrailleUtilities` that converts text into a “seeable” Braille representation using the following alphabet:
In the real Braille alphabet, the small dots are not actually present. We include them to make their absence more visible to seeing people. The larger dots appear as raised bumps on the display surface. The class has the following:

1. String constant RAISED defined to be \u2022, which represents the raised bump present in the real Braille alphabet. It is public, static, and final.

2. String constant UNRAISED defined to be \u00b7, which represents the artificial unraised bump that we have introduced in our Braille representation. It is public, static, and final.

3. String constant LETTER_SPACER defined to be two spaces. This padding will appear between letters to make the Braille letters easier to separate visually. It is public, static, and final.

4. String constant WORD_SPACER defined to be four spaces. This padding will appear between words and is therefore intentionally larger than LETTER_SPACER. It is public, static, and final.

5. Method translateTopLine, which accepts one parameter: the text to translate to Braille, of type String. It returns just the top line of the Braille translation. Process the text using the following procedure:

   (a) Replace each existing space character (which likely separates words) in the text with WORD_SPACER.

   (b) Lowercase all letters in the text. We will not distinguish cases in our representation.

   (c) Replace all lowercase case letters with the top lines of their Braille representation, with a LETTER_SPACER appearing after each letter. For example, each ‘a’ is replaced by a raised dot, an unraised dot, and a letter spacer. Each ‘b’ is replaced with the same thing. Each ‘c’ is replaced with two raised dots and a letter spacer.

   (d) Remove any leading or trailing space in the String. The trim method in the String is one way to accomplish this.

Hold on! Before writing 26 separate replace commands, observe that the top row can only be one of three sequences: raised-unraised, unraised-raised, or raised-raised. Save yourself considerable work by clustering the letters according to their Braille sequence. Then use String.replaceAll and some magic called a regular expression to process each of the three clusters very succinctly. Regular expressions let one describe patterns of text in a non-literal
way. To describe a character that is one of a fixed set, we use a regular expression with a
character class. For example, the expression [aeiou] matches any vowel. To replace each ‘a’
and ‘b’ en masse in text, we write text.replaceAll("[ab]", ...).

6. Method translateMiddleLine, which accepts one parameter: the text to translate to Braille,
of type String. It returns just the middle line of the Braille translation. Process the text
using a procedure similar to translateTopLine.

7. Method translateBottomLine, which accepts one parameter: the text to translate to Braille,
of type String. It returns just the bottom line of the Braille translation. Process the text
using a procedure similar to translateTopLine.

8. Method translate, which accepts one parameter: the text to translate to Braille, of type
String. It returns the Braille translation of the text, with a newline after each. Do not use
the linefeed character \n to separate lines, as this isn’t the correct way to separate lines on
every operating system. Instead, use %n with String.format to get the correct newline for
the operating system running your code. For example, String.format("a%n\b").

4 Implementation Hints

Don’t read this section—at least not right away. It will rob you of the chance to think hard, which
is the main ingredient of learning.

1. For NumberUtilities.round10, method Math.round will round to the nearest whole number,
which isn’t exactly what you want. But try dividing the number by 10 first...

2. For NumberUtilities.getGameCount, draw increasingly larger tournament brackets and count
the games. What do you notice to always be true about the number of games? How is the
number of games linked to the number of rounds?

3. For NumberUtilities.getFraction, the fractional portion of a positive number is the differ-
ence between the number and the nearest whole number less than or equal to it. There is a
method in the Math class that computes this nearest whole number and another that turns a
negative number positive.

4. For WebUtilities.getHost, use your standard String surgery methods indexOf and substring.
Finding the location of the second and third slashes may require a version of indexOf that
accepts multiple parameters.

5. For WebUtilities.getTitle, this one’s not that different than WebUtilities.getHost.

5 Submission

To submit your work for grading:

1. Put the SpecChecker for this homework in your Build Path. Run the SpecChecker as a Java
Application and fix problems until all tests pass.
2. Commit and push your work to your repository. Verify that your solution is on Bitbucket.

A passing SpecChecker does not guarantee you credit. Your grade is conditioned on a few things:

- You must meet the requirements described above. The SpecChecker checks some of them, but not all.

- You must not plagiarize. Write your own code. Talk about code with your classmates. Ask questions of your instructor or TA. Do not look at others' code. Do not ask questions specific to your homework anywhere online but Piazza. Your instructor employs a vast repertoire of tools to sniff out academic dishonesty, including: drones, CS 145 moles, and a piece of software called MOSS that rigorously compares your code to every other submission. You don’t want to live in a world serviced by those who squeaked by through questionable means. For your future self, career, and family, do your own work.

- Your code must be submitted correctly and on time. Most excuses devolve into, “I started too late.” The fix for this problem is not an extension.